

**UNIVERSIDAD DEL CEMA
Buenos Aires
Argentina**

Serie
DOCUMENTOS DE TRABAJO

Área: Ingeniería Informática

**ARQUITECTURA DE SOFTWARE ACADÉMICA
PARA LA COMPRENSIÓN DEL DESARROLLO
DE SOFTWARE EN CAPAS**

Darío G. Cardacci

**Octubre 2015
Nro. 574**

**www.cema.edu.ar/publicaciones/doc_trabajo.html
UCEMA: Av. Córdoba 374, C1054AAP Buenos Aires, Argentina
ISSN 1668-4575 (impreso), ISSN 1668-4583 (en línea)
Editor: Jorge M. Streb; asistente editorial: Valeria Dowding <jae@cema.edu.ar>**

ARQUITECTURA DE SOFTWARE ACADÉMICA PARA LA COMPRESIÓN DEL DESARROLLO DE SOFTWARE EN CAPAS

Ing. Darío G. Cardacci*

ABSTRACT

El desarrollo de software implica considerar una cantidad variada de aspectos tecnológicos. Entre los más destacados podemos mencionar los relacionados con: el acceso a datos, las interfaces, los procesos funcionales, el control de las transacciones, la accesibilidad y la seguridad. Lograr un diseño coherente con los requerimientos planteados, niveles aceptables de flexibilidad, extensibilidad y usabilidad, así como facilitar las actividades de mantenimiento (preventivo, adaptativo, correctivo, evolutivo y perfectivo) lleva a pensar la concepción del software sustentado en una arquitectura. Una de las formas arquitectónicas más conocidas propone dividir al software en capas, donde cada una representa un agrupamiento lógico que se corresponde con un elemento físico del sistema deseado. Las capas planteadas en el diseño arquitectónico deben tener la suficiente independencia para tratarlas como unidades funcionales independientes y las interfaces de sus componentes permitirán que interactúen de manera significativa y eficiente entre sí. Con el diseño en capas se intenta lograr que los cambios que se realizan en el sistema impacten de forma acotada en el lugar afectado y que los efectos no deseados en el resto del software disminuyan a su mínima cantidad. Este trabajo presenta un diseño arquitectónico que académicamente permite comprender con facilidad los aspectos planteados y establece un punto de partida para lograr arquitecturas más complejas.

* Las opiniones y/o comentarios que pudieran generarse en este trabajo son de exclusiva responsabilidad de su autor y no necesariamente expresan la posición de la Universidad del CEMA.

INTRODUCCIÓN

El desarrollo de software ha evolucionado y entre los aspectos a los que se le prestaron más atención encontramos el uso eficiente y adecuado de los recursos. Esto ha llevado a que la ingeniería de software trabajase arduamente en las actividades previas a la codificación. Entre estas, el diseño de la arquitectura de software ocupa un lugar peculiar, pues las consecuencias que genera trabajar sobre una arquitectura adecuada o una que no lo es, son muy importantes.

Cuando diseñamos la arquitectura de nuestro sistema lo hacemos pensando en que estamos construyendo el marco estructural global que le dará sustento a todo nuestro desarrollo. Como en cualquier aspecto de la vida, si lo estructural no es correcto, es muy probable que tengamos serios problemas al corto o largo plazo.

Al referirnos a arquitectura de software aludimos a *“la estructura general de este y a las formas en las que ésta da integridad conceptual a un sistema”* [Sha95].

Los trabajos realizados en este sentido le dan un nueva visión al diseño de software estableciendo que *“una meta del diseño de software es obtener una aproximación arquitectónica de un sistema”* [Pre10a].

Los aspectos arquitectónicos han tenido tal relevancia que IEEE en su “IEEE-std-1471-2000, Recommended Practice for Architectural Description of Software-Intensive System.” [IEE00] realiza una interesante propuesta que establece objetivos para la descripción arquitectónica que por razones de espacio no abordaremos en este momento.

Con el transcurso del tiempo se han elaborado múltiples formas arquitectónicas, siempre con el objetivo de separar conceptualmente los desafíos que nos presenta el desarrollo de software y de esta manera lograr que cada elemento sea lo suficientemente independiente como para que los cambios efectuados en uno de ellos no afecte al resto.

Entre las arquitecturas más destacadas podemos mencionar: arquitecturas centradas en los datos, arquitecturas de flujos de datos, arquitecturas de call-back, arquitecturas cliente-servidor, arquitecturas orientadas a objetos, arquitecturas basadas en componentes, arquitecturas orientadas a servicios (SOA¹), la arquitectura modelo, vista controlador (MVC²) y la arquitectura en capas.

Para ampliar los detalles de algunas arquitecturas, Roger Pressman en su libro *“Ingeniería de Software. Un enfoque práctico”* [Pre10b] realiza un recorrido y explica las características de algunas de ellas.

¹ **Service Oriented Architecture.** Es un paradigma para diseñar y desarrollar sistemas distribuidos. Las soluciones SOA han sido creadas para poder integrar con facilidad y flexibilidad otros sistemas y exponer de manera significativa los servicios ofrecidos por el SI para su invocación. Permite crear sistemas de información altamente escalables que reflejan el negocio de la organización.

² **Model View Controller.** Patrón de arquitectura de software que separa los datos, la lógica del negocio, la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador. MVC fue introducido por Trygve Reenskaug en Smalltalk-76. En los años 80, Jim Althoff y otros implementaron una versión de MVC para la biblioteca de clases de Smalltalk. Sólo más tarde, en 1988, MVC se expresó como un concepto general en un artículo sobre Smalltalk-80.

En la actualidad lo planteado ha cobrado notoriedad al punto que desde hace un tiempo es normal escuchar hablar sobre el rol del arquitecto de software y sus aportes al producto final.

De por sí es un gran desafío construir arquitecturas eficientes, pero lo es mucho más tratar de explicarles a los nóveles cómo funcionan y cuál es su rol.

El presente trabajo plantea una arquitectura sencilla inspirada en la arquitectura en capas, que cubre los aspectos necesarios para comprender las ventajas de utilizarla y adoptarla como punto de partida para el desarrollo de arquitecturas más complejas.

ARQUITECTURA EN CAPAS

La arquitectura en capas es una de la más fuertemente adoptadas por los desarrolladores. Una capa representa un aspecto lógico a tratar como una unidad propia con sus objetivos funcionales claramente definidos, sus dependencias y sus colaboraciones hacia el sistema. Se pueden encontrar arquitecturas en capas de dos capas, tres capas y n capas. En este contexto definiremos nivel como la representación física de una capa. Por ejemplo, una capa del diseño arquitectónico que desarrollemos podría estar representada físicamente por un proyecto en particular. No existe ninguna restricción en que la relación sea uno a uno entre capa y nivel. No obstante, suele producirse ya que las fronteras establecidas físicamente en los niveles acentúan la clasificación lógica que diseñamos en las capas de nuestra arquitectura.

Pensemos por un momento en un sistema extremadamente sencillo que proporciona la funcionalidad CRUD³. Si observamos cada una de las actividades que se nos plantean (agregar, borrar, modificar y consultar) cada una de ellas potencialmente podría involucrar tres tipo de funcionalidades muy distintas. Estas son:

- Las relacionadas con el usuario. La manera en que el SI⁴ interactúa con él, la captura de los datos ingresados y su visualización como resultado de consultas o cualquier otra operación que implique la recuperación de datos.
- Las reglas lógicas. Son las que afectan a los datos ingresados generando nueva información a partir de ellos. También se incluyen aquí los datos calculados sobre los datos almacenados, cuando existe un requerimiento de visualización.
- La persistencia de los datos. Se refiere al tratamiento de los estados de los objetos para ser almacenados físicamente en las bases de datos.

Surge de lo expuesto que estas tres funcionalidades plantean problemas lo suficientemente peculiares

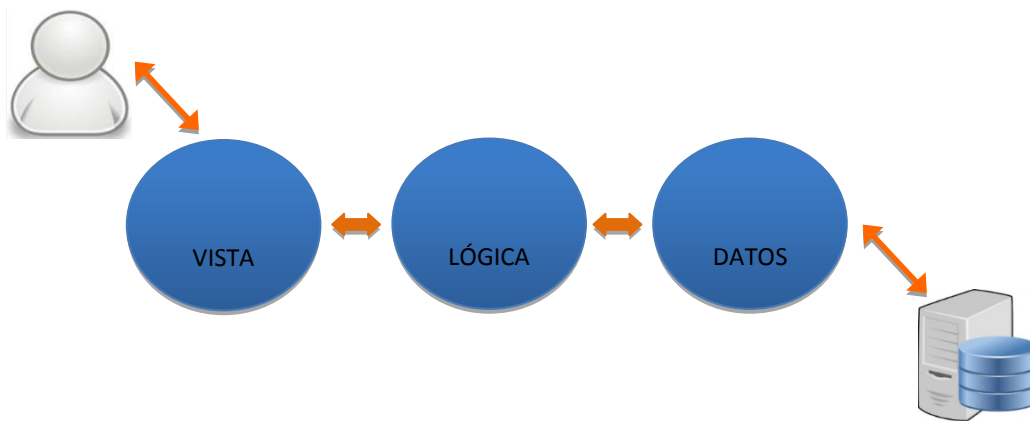
³ CRUD: Significa Create, Read, Update y Delete en los SI.

⁴ SI. Sistema de Información

como para ser tratados por separado. Considerando esto, podemos plantear una arquitectura de tres capas donde se denominará respectivamente a cada una:

- Capa de Vista
- Capa de Lógica
- Capa de Datos

Un esquema representativo de lo expuesto sería:



Si tomamos como ejemplo implementar la funcionalidad de agregar, borrar, modificar y consultar clientes, para lograrlo tendríamos que construir en nuestra arquitectura tres clases que darán origen a sus respectivos objetos.

La primera podemos denominarla *“ClienteVista”*. En ella se colocará todo aquello que permite controlar los aspectos que estén relacionados con la interfaz gráfica del usuario, así como lo necesario para interpretar las acciones que este solicite y las formas de mostrarle la información. Por ejemplo si el usuario solicita grabar los datos ingresados en la GUI, este objeto sería el encargado de traducir esa petición en un mensaje concreto al objeto que corresponda de la capa de lógica, además de proveerle la referencia al objeto que posee el estado a persistir. Si por el contrario la solicitud del usuario es una consulta, *“ClienteVista” transformará este requerimiento en un mensaje al objeto indicado de la capa de lógica, conjuntamente con una referencia al objeto que posee el código a consultar y permanecerá (esto también puede construirse de forma asincrónica) a la espera del arribo de los datos consultados para presentarlos adecuadamente en la GUI.*

La segunda la denominamos *“ClienteLogica”*. En esta capa se definirán las clases que modelan la solución que implementamos y la lógica funcional, en nuestro caso referida al cliente. En este peculiar ejemplo quizá resulte confuso identificar qué reglas lógicas o funcionales pueden estar dadas al hacer un alta de cliente. Esto es normal, en general en las funcionalidades CRUD no abundan, pero serán

fácilmente identificables cuando esto se aplique a clases que intervienen en los genuinos procesos funcionales del sistema. También desde aquí se emiten los mensajes hacia la capa de datos para que se concreten las operaciones de acceso a los datos almacenados y la referencia hacia el objeto que posee en su estado los datos a ser tratados.

La tercera se llamará “ClienteDato”. La funcionalidad que debe cubrir abarca la descomposición sistemática del estado del objeto que posee los datos, a estructuras manipulables en los términos del modelo de acceso a datos utilizado y la base de datos seleccionada para persistirlos. En definitiva debe oficiar como un ORM⁵. También se encargará de gestionar las conexiones y ejecución de comandos necesarios para operar sobre la base de datos.

Con lo expuesto se puede observar claramente la separación funcional de los aspectos relacionados con el manejo de las vistas, la lógica y los datos. No obstante este enfoque puede superarse logrando capas más refinadas en una nueva arquitectura que le dará mayor flexibilidad al diseño así como una mejor separación de los conceptos.

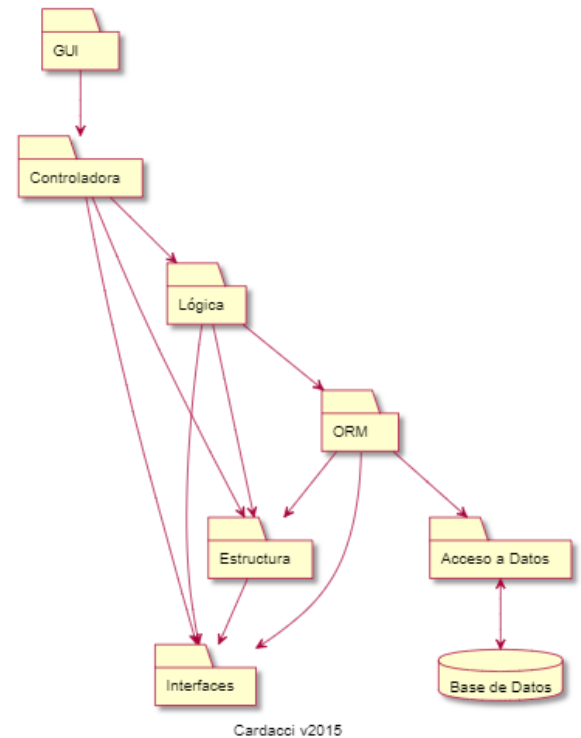
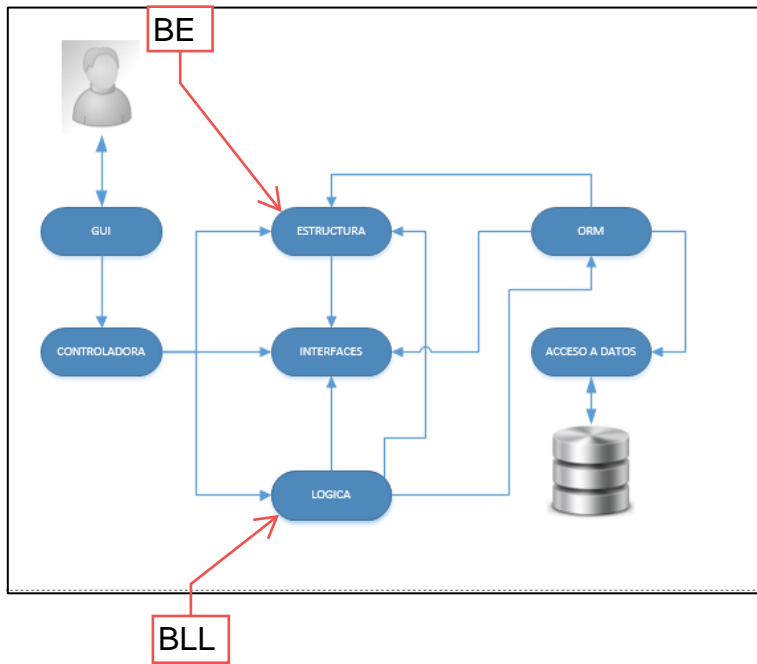
ARQUITECTURA N-CAPAS

La arquitectura que proponemos para comprender estos conceptos posee 7 capas denominadas:

- Capa GUI
- Capa Controladora
- Capa de Interfaces
- Capa Estructura
- Capa Lógica
- Capa ORM
- Capa Acceso a Datos

El esquema muestra la arquitectura completa con sus dependencias. Para su desarrollo se ha utilizado Visual Studio 2013, SQL 2014, Visual Basic.NET y ADO.NET. Con muy poco esfuerzo puede trasladarse a otra plataforma o lenguaje. La arquitectura está contenida en una solución y cada capa construida como un proyecto independiente dentro de la solución. Las flechas de líneas continuas indican que el proyecto referencia al apuntado.

⁵ ORM: **Mapeo Objeto-Relacional** (Object-Relational Mapping). Es una técnica de programación para convertir datos entre el SI desarrollado en un lenguaje de programación orientado a objetos y lograr la persistencia de sus estados en una base de datos relacional.



A continuación se desarrollará la arquitectura capa por capa mostrando su funcionalidad e integración. Por razones de espacio y exposición algunas piezas de código se presentan parcialmente.

CAPA GUI

La capa GUI⁶ contiene todas las interfaces gráficas que interactúan con el usuario considerando el flujo bidireccional de datos (los ingresados por el usuario y los entregados a él por el SI). Estas clases definen la manera en que el usuario interactúa con el SI. Algunos diseñadores optan por colocar aquí validaciones elementales que se pueden resolver a nivel de GUI sin que intervengan reglas del negocio. Por ejemplo, validar que en un campo numérico no se ingrese texto o que en un campo de fecha la misma tenga un formato adecuado. La ventaja de colocar este tipo de validaciones en esta capa es que no se sobrecarga la estructura con mensajes hacia otras. Esto es muy positivo en sistemas distribuidos y sistemas habilitados para la web, ya que si estas validaciones básicas se realizan en un servidor o más específicamente en una capa que se encuentra en un servidor, implicaría un flujo de datos adicional que en casos extremos puede atentar contra la performance global del sistema. Para nuestro usaremos una GUI para manejar el CRUD de clientes y esta GUI bien podría ser un objeto de tipo "Form". Esta GUI será controlada por un controlador en particular donde se colocará la lógica que interpreta todas las acciones realizadas por el usuario en la GUI. La capa GUI posee una referencia hacia la capa controladora, esto permite instanciar un objeto controlador en la GUI y como el constructor del objeto controlador acepta objetos GUI, este se puede pasar como parámetro. Veamos esto en código:

Un "controlador en particular" implica MPV

⁶ GUI. Interfaz Gráfica de Usuario, elemento de un programa informático que actúa de interfaz con el usuario utilizando un conjunto de íconos, imágenes y objetos gráficos para representar la información y acciones disponibles en ella.

El objeto GUI es el “Form1” y la clase “ClienteVista” la controladora de “Form1”.


```
Public Class Form1
    Dim ControladorCliente As New ClienteVista(Me)
End Class
```

Lo único que resta colocar en la GUI son los llamados a las acciones que puede resolver la controladora. Como ejemplo supongamos que en nuestra GUI tenemos un botón llamado “button2” del tipo “button”. Sobre él se opera para lograr un alta de cliente. El código resultante es:

```
Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button2.Click
    ControladorCliente.Alta()
End Sub
```

Como se puede observar las GUI quedan bastantes despojadas y solo contendrán las validaciones primarias que comentamos oportunamente y las invocaciones a la controladora. Esto permite que esta capa sea intercambiable fácilmente por otros tipos de formularios. También podemos hacer que el usuario obtenga el contenido de un texto en su versión oral realizando cambios mínimos. Si la salida es hacia la web, dependiendo como lo hemos diseñado, bastará con ajustar solo esta capa o ella y su controladora asociada. De todas maneras el impacto será mínimo respecto del sistema global.

CAPA CONTROLADORA

La capa controladora desempeña un rol fundamental pues oficia como un proxy  entre las GUI, las estructuras de clases del negocio y los comportamientos. Además implementa una interfaz denominada “IABMC” donde se estandarizan y definen las acciones comunes al CRUD y que no son operaciones referidas a las reglas de negocio.

Antes de adentrarnos en la funcionalidad de la controladora cabe aclarar que, en una decisión de diseño cada concepto del negocio se encuentra dividido en dos clases que pertenecen a dos capas distintas. Por un lado lo que se denomina “estructura”, que como su nombre lo expresa representa la estructura de una entidad del negocio. En el caso de los clientes podría ser sus características (id, nombre, fecha de alta, activo), inclusive aquellos elementos como los agregados estructuralmente, por ejemplo sus teléfonos. Por otro lado la capa denominada “lógica”, que contiene los comportamientos correspondiente a las entidades del negocio (reglas de negocio), incluyendo los definidos por las operaciones básicas del CRUD. Esta decisión de diseño permite recombinar dinámicamente una estructura con más de un comportamiento o bien reutilizar un comportamiento estándar en varias estructuras. Sería muy extenso plantear las ventajas y desventajas de hacer esto en el presente trabajo, simplemente se adoptó esta forma pues al ser una arquitectura académica se ha deseado

demostrar la posibilidad de hacerlo. Tan solo dejaré expresado que al utilizar esta propuesta se simplifican notablemente las relaciones jerárquicas del tipo especialización – generalización (herencia).

Habiendo inevitablemente aclarado lo expresado en los párrafos anteriores, pasaremos a ver la taxonomía interna de las controladoras. Para tener un hilo conductor seguiremos con el ejemplo de cliente.

Primariamente una controladora agrega una estructura y un comportamiento para una entidad del negocio. En nuestro ejemplo la controladora de cliente agrega una estructura de cliente y un comportamiento para este cliente. Las clases se denominan “*Cliente*” para la estructura y “*ClienteD*” para el comportamiento.

A continuación se presenta una visión parcial de la clase controladora “*ClienteVista*” donde se puede observar la agregación mencionada y el puntero a la GUI por medio del campo *Vgui*. Esta capa posee tres referencias, la primera hacia la capa estructura, la segunda hacia la capa lógica y la tercera hacia la capa interfaces. La clase “*ClienteVista*” completa, también implementa las definiciones realizadas en la interfaz “*IABMC*”. Como esta interfaz es genérica, esta capa aprovecha la referencia a la capa estructura para tener acceso al tipo “*Cliente*” y así tipar la interfaz.

```
Public Class ClienteVista
```

```
    Implements ABMC_Interfaces.IABMC(Of Cliente)
```

```
    Private Vcliente As Cliente
```

```
    Private VclienteD As ClienteD
```

```
    Private Vgui As Form
```

```
Sub New(pGUI As Form)
```

```
    Me.Cliente = New Cliente
```

```
    Me.ClienteD = New ClienteD
```

```
    Me.Vgui = pGUI
```

```
End Sub
```

```
Public Property Cliente() As Cliente
```

```
    Get
```

```
        Return Vcliente
```

```
    End Get
```

```
Set(ByVal value As Cliente)
```

```
    Vcliente = value
```

```
End Set
```

```

End Property

Private VclienteD As ClienteD

Public Property ClienteD() As ClienteD

    Get

        Return VclienteD

    End Get

    Set(ByVal value As ClienteD)

        VclienteD = value

    End Set

End Property

End Class

```

Veamos a continuación el caso de alta del cliente. La controladora toma los datos de la GUI y los coloca en la estructura del cliente, alterando su estado y dejándolo con los valores que se deben dar de alta. En nuestro caso el cliente agrega teléfonos y se tomó la decisión de diseño que sea la controladora de cliente la que posea este conocimiento y gestione la incorporación de ellos al cliente. Otra forma de realizar esto es crear una controladora de teléfonos. En este caso la controladora de cliente le solicitará servicios a esta. Por razones de complejidad no se adoptó esta última propuesta, pero sin duda sería más acorde, pues si el tratamiento de teléfonos es el mismo para distintas entidades como: cliente, proveedor, empleado etc. se reutilizaría el servicio de esa controladora. Finalmente se le envía un mensaje de “Alta” a la capa lógica por medio del objeto agregado “*ClienteD*”. En esa capa se resuelvan la reglas de negocio si existieran y se prosige con el proceso de alta. Es importante destacar que el mensaje lleva un parámetro que es el puntero al objeto estructura que es accedido por medio de la propiedad “*Cliente*”. Este objeto contiene los valores a ser dados de alta.

```

Public Sub Alta(Optional ByRef QueObjeto As Cliente = Nothing) Implements IABMC(Of Cliente).Alta

    With Me.Cliente

        .Id = Integer.Parse(Vgui.Controls.Find("Id", False).First.Text)

        .Nombre = Vgui.Controls.Find("Nombre", False).First.Text

        .FechaAlta = Date.Parse(Vgui.Controls.Find("FechaAlta", False).First.Text)

        .Activo = If(Vgui.Controls.Find("Activo", False).First.ToString.Last = "1", True, False)

        .Telefonos.Clear()

        Do While If(MsgBox("¿Desea ingresar un teléfono?", MsgBoxStyle.YesNo) = MsgBoxResult.Yes, True, False)

```

```

        .Telefonos.Add(New Telefono(Nothing, InputBox("Ingrese el número de teléfono")))

Loop

'Le envía un mensaje a Lógica

Me.ClienteD.Alta(Me.Cliente)

End With

End Sub

```

CAPA INTERFACES

En esta capa se definen las interfaces del sistema. Para el ejemplo existen dos. La primera se denomina “IABMC” y la segunda “IID”. El objetivo de la interfaz “IABMC” es definir las acciones comunes que propone el CRUD y de esta forma lograr que la mensajería sea estándar entre las capas controladora, lógica y datos así como en las distintas clases que se agreguen al SI.

Las acciones definidas en la interfaz “IABMC” son:

```

Public Interface IABMC(Of T)

    Sub Alta(Optional ByRef QueObjeto As T = Nothing)

    Sub Baja(Optional ByRef QueObjeto As T = Nothing)

    Sub Modificacion(Optional ByRef QueObjeto As T = Nothing)

    Function Consulta(Optional ByRef QueObjeto As T = Nothing) As List(Of T)

    Function ConsultaRango(Optional ByRef QueObjeto1 As T = Nothing, Optional ByRef QueObjeto2 As T = Nothing) As List(Of T)

End Interface

```

La interfaz es genérica (of T) esto permite que los parámetros de las acciones adopten el tipo que se les pasa al momento de implementarla. Esto causa que cuando se implementa en cliente los parámetros sean de ese tipo y esto se irá adecuando al tipo de objeto donde la implementemos. La ventaja que obtenemos de hacerlo así y no utilizando un parámetro del tipo Object es que evitamos el cambio de tipo del objeto al cual apunta el parámetro de la acción.

La segunda interfaz “IID” define solo una funcionalidad. Es utilizada para estandarizar la manera en que se retorna el *Id* (identificador). Opera como un decorador logrando de esta manera que en capas posteriores (capa de datos) la consulta del *Id* sea estándar. ???

```

Public Interface IID
    Function RetornaId() As Integer
End Interface

public interface IID
{
    int RetornaId();
}

```

CAPA ESTRUCTURA

Esta capa agrupa todas las estructuras de las entidades que nuestro SI posea. En nuestro ejemplo las clases “Cliente” y “Teléfono”. Estas clases implementan **clonación profunda** y la interfaz “IID”. En el caso particular de “*Cliente*”, además de las características propias implementa la **agregación** de “*Teléfono*”

Clase CLIENTE

Public Class Cliente

Implements ICloneable

Implements ABMC_Interfaces.IID

Private vTelefonos As New List(Of Telefono)

Public Property Telefonos() As List(Of Telefono)

Public Property Id() As Integer

Public Property Nombre() As String

Public Property FechaAlta() As Date

Public Property Activo() As Boolean

Public Function Clone() As Object Implements System.ICloneable.Clone

Dim RetornoCliente As Cliente = CType(Me.MemberwiseClone, Cliente)

If Not Me.Telefonos Is Nothing Then

For Each T As Telefono In Me.Telefonos

RetornoCliente.Telefonos.Add(T.Clone)

Next

End If

Return RetornoCliente

End Function

Sub New()

End Sub

Sub New(ByVal Queld As Integer, ByVal QueNombre As String, ByVal QueFechaAlta As Date, ByVal QueActivo As Boolean)

Me.Id = Queld

Me.Nombre = QueNombre

```

        Me.FechaAlta = QueFechaAlta

        Me.Activo = QueActivo

    End Sub

    Public Function Retornald() As Integer Implements IID.Retornald

        Return Me.Id

    End Function

End Class

```

Clase TELEFONO

```

Public Class Telefono

    Implements ICloneable

    Implements IID

    Public Property Id() As Integer

    Public Property Numero() As String

    Sub New()

        End Sub

    Sub New(ByVal Queld As Integer, ByVal QueNumero As String)

        Me.Id = Queld

        Me.Numero = QueNumero

    End Sub

    Public Function Clone() As Object Implements System.ICloneable.Clone

        Return Me.MemberwiseClone

    End Function

    Public Function Retornald() As Integer Implements IID.Retornald

        Return Me.Id

    End Function

End Class

```

CAPA LOGICA

Esta capa contiene las clases que representan el comportamiento de las entidades de negocio. Aquí se definen las reglas de negocio del SI. Se debe lograr una asignación adecuada de las reglas a las clases que corresponda según su responsabilidad. Es importante aclarar que por la simplicidad del ejemplo planteado (básicamente un CRUD) podría parecer que las clases de esta capa ofician como simples pasarelas. En la realidad esto no es así. Todas las operaciones sobre los estados de los objetos de nuestro SI actúan a este nivel. Frecuentemente, una manera sencilla de analizar la complejidad funcional de un SI es observando las reglas que se ofrecen en esta capa. La capa lógica posee una referencia a la capa estructura, esto es debido a que al implementar la interfaz “IABMC” que es genérica, debe tener acceso al tipo “Cliente”. Por otro lado la referencia a la capa ORM es para poder enviarle un mensaje acorde a la acción que esté tratando de resolver. Como las clases de la capa lógica y la capa ORM implementan la Interfaz “IABMC”, existe un paralelismo entre la acción que se está ejecutando en la capa lógica (por ejemplo “Alta”) y el mensaje que se estará enviando a la clase mapeadora de la capa ORM.

Una visión parcial de la clase “*ClienteD*” correspondiente a la capa lógica, permite observar claramente la agregación del objeto “*ClienteDatos*” de la capa orm y el mensaje hacia el mismo `Me.ClienteDatos.Alta(QueObjeto)`

```
Public Class ClienteD
```

```
    Implements ABMC_Interfaces.IABMC(Of Cliente)
```

```
    Private VclienteDatos As ClienteDatos
```

```
    Public Property ClienteDatos() As ClienteDatos
```

```
        Get
```

```
            Return VclienteDatos
```

```
        End Get
```

```
        Set(ByVal value As ClienteDatos)
```

```
            VclienteDatos = value
```

```
        End Set
```

```
    End Property
```

```
    Sub New()
```

```
        Me.ClienteDatos = New ClienteDatos
```

```
    End Sub
```

```
Public Sub Alta(Optional ByRef QueObjeto As Cliente = Nothing) Implements IABMC(Of Cliente).Alta
```

```
Me.ClienteDatos.Alta(QueObjeto)
```

```
End Sub
```

```
End Class
```

CAPA ORM

Esta capa cumple el rol de tomar el objeto cuyo estado contiene los valores a persistir y descomponerlo en términos del modelo de acceso a datos utilizado y las reglas planteadas por las bases de datos relacionales. **Esto es necesario pues las reglas que rigen para la orientación a objetos no siempre coinciden con las reglas relacionales.** Por ejemplo, en el plano de los objetos es correcto que el objeto “*Cliente*” agregue una colección de teléfonos, sin embargo en la base de datos podríamos tener dos tablas: cliente y teléfono, o tres tablas: cliente, teléfono y la asociativa entre ellas, según el tipo de relación que se plantee. **La distribución de los valores que posee el objeto en su estado de acuerdo a lo que corresponda, es la tarea que realiza la clase que denominamos mapeadora y se encuentra en esta capa.**

Existen dos maneras que la clase mapeadora actúe sobre la BD, una es invocando el/los **procedimientos almacenados** que se construyen con este fin. En este caso los datos constituirán la colección de parámetros para el procedimiento utilizado. La segunda manera es generando el **comando SQL** y solicitándole al objetos del la capa de datos que lo ejecute. Por simplicidad utilizamos esta última.

En esta capa también se realiza el trabajo inverso cuando la operación solicitada así lo requiera. Por ejemplo si fuera una consulta de un cliente, se debe tomar el objeto de datos que se obtiene (**un datatable en nuestro caso por utilizar ADO.NET**) y componerlo como **un objeto con su estado (cliente)** o **una colección de objetos según corresponda (teléfonos).**

En nuestro peculiar ejemplo actúan dos clases del orm, la correspondiente a los clientes (“*ClienteDatos*”) y la correspondiente a los teléfonos (“*TelefonoDatos*”). La primera le solicita colaboración a la segunda como se verá a continuación. La capa orm posee referencia a la capa estructura y a la capa acceso a datos. La primera referencia permite que las clases de esta capa, al implementar la interfaz “*IABMC*” que es genérica, tengan disponible los tipos del modelo de negocio disponible en la capa estructura. En nuestro ejemplo los tipo “*Cliente*” y “*Telefono*”. La segunda referencia nos da acceso a los servicios de acceso físico a los datos.

```
Public Class ClienteDatos
```

```
Implements ABMC_Interfaces.IABMC(Of Cliente)
```

```
Dim VtelefonoDatos As New TelefonoDatos
```



```

Public Sub Alta(Optional ByRef QueObjeto As Cliente = Nothing) Implements IABMC(Of Cliente).Alta
    Try
        Dim dt As DataTable = Comando.ObjStructureTable("cliente")
        Dim dr As DataRow = dt.NewRow
        With QueObjeto
            dr.ItemArray = {.Id, .Nombre, .FechaAlta, .Activo}
            dt.Rows.Add(dr)
            Comando.ActualizaBase("cliente", dt)
            If .Telefonos.Count > 0 Then
                VtelefonoDatos.ObjetoSolicitante = QueObjeto
                For Each Tr As Telefono In .Telefonos
                    VtelefonoDatos.Alta(Tr)
                Next
            End If
        End With
        Catch ex As Exception
            MsgBox("Error en el Alta")
        End Try
    End Sub
End Class

```

En el código anterior se puede observar en el mapeador de cliente la agregación del mapeador de teléfonos por medio del objeto “*VtelefonoDatos*”. Este último le dará servicios cuando llegue el momento de descomponer los datos del cliente. Luego el procedimiento “*Alta*” que es parte de la implementación propuesta por la interfaz “*IABMC*” recibe el objeto estructura con los datos en su estado. Desde esta capa se le solicita servicios a la capa acceso a datos. Por medio de *Comando.ObjStructureTable("cliente")*, se obtiene un datatable denominado “*dt*” cuya estructura responde a la tabla “*Cliente*” de la base de datos. Con *Dim dr As DataRow = dt.NewRow* se obtiene un nuevo datarow y en el se cargan los datos que portaba el estado del objeto apuntado por “*QueObjeto*” que llegó como parámetro del procedimiento “*Alta*”. Esto se logra con la instrucción *dr.ItemArray = {.Id, .Nombre, .FechaAlta, .Activo}*. Por medio de *dt.Rows.Add(dr)* se agrega el datarow al datatable “*dt*”. Luego se actualiza la tabla “*cliente*” de la base de datos con *Comando.ActualizaBase("cliente", dt)*. Como un cliente puede agregar teléfonos se evalúa si esto es así, en caso afirmativo por medio de *VtelefonoDatos.ObjetoSolicitante = QueObjeto* se le informa al mapeador de teléfonos cual es el objeto que solicita el servicio. Esto es

necesario porque dicho mapeador tendrá que persistir los teléfonos y la clave foránea en esta tabla. Para lograrlo necesita conocer el Id del cliente al cual pertenecen los teléfonos. El siguiente paso es recorrer cada teléfono y solicitarle el servicio al mapeador de teléfonos con *VtelefonoDatos.Alta(Tr)*.

El código parcial del mapeador de teléfono es:

```
Public Class TelefonoDatos

    Implements ABMC_Interfaces.IABMC(Of Telefono)

    Private VobjetoSolicitante As IID

    Public WriteOnly Property ObjetoSolicitante() As IID

        Set(ByVal value As IID)

            VobjetoSolicitante = value

        End Set

    End Property

    Public Sub Alta(Optional ByRef QueObjeto As Telefono = Nothing) Implements IABMC(Of Telefono).Alta

        Dim dt1 As DataTable = Comando.ObjStructureTable("telefono")

        Dim contador As Integer = Comando.ObjDataTable("select max(telid) from telefono").Rows(0).Item(0) + 1

        Dim dr1 As DataRow = dt1.NewRow

        dr1.ItemArray = {contador, QueObjeto.Numero, Me.VobjetoSolicitante.Retornald}

        dt1.Rows.Add(dr1)

        contador += 1

        Comando.ActualizaBase("telefono", dt1)

    End Sub

End Class
```

Podemos observar que el código anterior realiza un trabajo similar al de cliente. Primero obtiene la estructura de la tabla teléfono con *Dim dt1 As DataTable = Comando.ObjStructureTable("telefono")*. Luego se obtiene un id de teléfono con la instrucción *Dim contador As Integer = Comando.ObjDataTable("select max(telid) from telefono").Rows(0).Item(0) + 1*. Lo siguiente es generar un nuevo datarow utilizando *Dim dr1 As DataRow = dt1.NewRow*. Se le cargan los datos *dr1.ItemArray = {contador, QueObjeto.Numero, Me.VobjetoSolicitante.Retornald}* y se agrega al datatable "*dt1*" con *dt1.Rows.Add(dr1)*. Finalmente se actualiza la base de datos usando *Comando.ActualizaBase("telefono", dt1)*.

CAPA ACCESO A DATOS

La capa de acceso a datos brinda estos servicios encapsulando y simplificando las posibilidades que brinda ADO.NET. Básicamente se sustenta en dos clases, "*Comando*" y "*Conexión*". Ambas

implementan SQLClient. La clase “Comando” brinda tres funcionalidades: a) retorna un datatable de acuerdo al sql que recibe, b) retorna la estructura de un datatable (sin datos) considerando el nombre de la tabla que recibe y c) Actualiza la BD.

La clase “Conexión” establece la conexión a la base de datos tomando los valores por defecto que posee o bien adoptando los que se le envían por parámetro.

```
Public Class Comando
```

```
    Private Shared Vcomando As SqlCommand
```

```
    Private Shared Function ObjComando(ByVal SelectCommand As String, ByVal Conexion As SqlConnection) As SqlCommand
```

```
        Vcomando = New SqlCommand
```

```
        Vcomando.CommandText = SelectCommand
```

```
        Vcomando.CommandType = CommandType.Text
```

```
        Vcomando.Connection = Conexion
```

```
        Return Vcomando
```

```
    End Function
```

```
    Shared Function ObjDataTable(ByVal SelectCommand As String) As DataTable
```

```
        Dim da As New SqlDataAdapter(ObjComando(SelectCommand, Conexion.ObjConexion))
```

```
        Dim dt As New DataTable
```

```
        da.Fill(dt)
```

```
        Return dt
```

```
    End Function
```

```
    Shared Sub ActualizaBase(ByVal TableName As String, ByVal dt As DataTable)
```

```
        Dim da As New SqlDataAdapter(ObjComando("Select * from " & TableName, Conexion.ObjConexion))
```

```
        Dim cb As New SqlCommandBuilder(da)
```

```
        da.InsertCommand = cb.GetInsertCommand
```

```
        da.DeleteCommand = cb.GetDeleteCommand
```

```
        da.UpdateCommand = cb.GetUpdateCommand
```

```
        da.Update(dt)
```

```
    End Sub
```

```
    Shared Function ObjStructureTable(ByVal TableName As String) As DataTable
```

```
        Dim da As New SqlDataAdapter(ObjComando("Select * from " & TableName, Conexion.ObjConexion))
```

```
        Dim dt As New DataTable
```

```
        da.FillSchema(dt, SchemaType.Mapped)
```

```

Return dt
End Function

End Class

Public Class Conexion

Private Shared VobjConexion As SqlConnection

Shared Function ObjConexion() As SqlConnection

VobjConexion = New SqlConnection("Data Source=.;Initial Catalog=GESTION;Integrated Security=True")

Return VobjConexion

End Function

Shared Function ObjConexion(ByVal QueStringDeConexion As String) As SqlConnection

VobjConexion = New SqlConnection(QueStringDeConexion)

Return VobjConexion

End Function

End Class

```

CONCLUSIONES

Luego de lo expuesto las funcionalidades han quedado lo suficientemente aisladas como para que los cambios realizados en cualquiera de ellas no impacten sobre el resto. Esto es muy favorable debido a que cualquier tipo de error generado no se propagará masivamente en el SI. A modo de ejemplo podemos mencionar que cambios en la estructura o el comportamiento del negocio no afectarán aspectos referidos con el control o presentación de las vistas y mucho menos con el acceso a datos. El mismo efecto causaría cualquier cambio en el resto de las capas.

La adopción de cualquier arquitectura implica que el equipo de diseñadores se comprometa con ciertos contratos. En nuestro caso este contrato implica una pequeña y aceptable sobrecarga de trabajo. Incorporar un concepto al negocio, implica considera su interfaz GUI, la controladora, su estructura, su comportamiento, y su clase mapeadora. Esto nos garantiza una forma controlada de crecimiento del SI. También otorga independencia en los aspectos estructurales, como he mencionado a lo largo del trabajo y el la dimensión funcional. **Particularmente este tipo de arquitectura se adapta mejor a sistemas cuyo tamaño y complejidad ameriten respetar los contratos establecidos. En caso contrario la sensación que causará es la de tener una sobrecarga de trabajo que no aporta demasiado.** Al ser presentada como una arquitectura académica para la comprensión primaria de algunos elementos a considerar al momento de trabajar con arquitecturas de SW, se ha dejado de lado una cantidad apreciable de factores a favor de alentar la simplicidad, pero los mismos podrían conformar otros trabajos per se. A continuación se enumeran algunos de ellos: a) Automatizar la forma en que trabajan los mapeadores. b) Incorporar una capa para el control de transacciones. c)

Incorporar un servicio para la obtención de las tablas que permiten persistir los estados de las clases de la capa de estructura. Seguramente esta lista podría ser bastante más larga. Dejamos esa tarea para el aporte que hagan los lectores. La idea final que deseo finalice este trabajo es que *“la utilización de arquitecturas de SW se torna inevitable si se desean obtener sistemas fácilmente escalables, flexibles y que su mantenimiento se pueda planificar y sea sustentable en el tiempo”*.

REFERENCIAS

- [IEE00] IEEE Standard Association, IEEE Standard 1471-2000 - IEEE Recommended Practice for Architectural Description for Software-Intensive Systems.
<http://standards.ieee.org/findstds/standard/1471-2000.html>
- [Pre10a] Pressman Roger S. (2010). Ingeniería del Software. Un enfoque práctico. 7ma. Edición. P(191). Mc Graw Hill. México.
- [Pre10b] Pressman Roger S. (2010). Ingeniería del Software. Un enfoque práctico. 7ma. Edición. Cap(9). P(206-233). Mc Graw Hill. México.
- [Sha05] Shaw, M. y Garlan, D. “Formulations and Formalisms in Software Architecture“, Volume 1000 - Lecture Notes in Computers Sciens, Springer-Verlag, 1995.

BIBLIOGRAFÍA

- Braude E.J. (2003) *Ingeniería de software - Una perspectiva orientada a objetos*. Editorial Alfaomega, México.
- Booch G. Cardacci D. (2013) Orientación a Objetos. Teoría y Práctica. Editorial Pearson Education. Buenos Aires. Argentina.
- Gamma E. Helm R. Johnson R. Vlissides J. (2003) Patrones de diseño. Editorial Addison Wesley, Pearson Education. Madrid.
- Meyer B. (1999) Construcción de software orientado a objetos. Editorial Prentice Hall. Madrid. España.
- Newkirk J.W. Vorontsov A.A. (2004) Test-Driven Development in Microsoft.NET. Editorial Microsoft Press. Redmond. Washington. EEUU.
- Pfleeger S.L. (2002) *Ingeniería de software - Teoría y práctica*. Editorial Prentice Hall, Argentina.
- Pressman R.S. (2010) *Ingeniería de software - Un enfoque práctico*. Séptima edición. Editorial McGraw Hill. México.
- Sanchez S. Sicilia M.A. Rodriguez D. (2012) Ingeniería de software. Un enfoque desde la guía SWEBOK. Editorial Alfaomega. México.
- Schach A.R. (2006) *Ingeniería de software clásica y orientada a objetos*. Editorial McGraw Hill, México.

- Sommerville I. (2005) *Ingeniería de software*. Editorial Pearson Addison Wesley, España.
- Weitzenfeld A. (2005) *Ingeniería de software orientada a objetos con UML Java e Internet*. Editorial Thomson, México.
- West D. (2004) *Object Thinking*. Editorial Microsoft Press. Redmond. Washington. EEUU.
- Wieggers K. (2003) *Software Requirements*. 2da ed. Editorial Microsoft Press. Redmond. Washington. EEUU.
- Withall S. (2007) *Software Requirement Patterns*. Editorial Microsoft Press. Redmond. Washington. EEUU.